A linear logic semantics for object-oriented, deductive and active databases

M.Bozzano G.Delzanno M.Martelli
DISI - Università di Genova,
Via Dodecaneso 35, I-16146 Genova, Italy
e-mail: bozzano@educ.disi.unige.it,
{giorgio,martelli}@disi.unige.it
phone: +39-10-3536727 (Martelli)

Abstract

Girard's linear logic [18] provides powerful means for studying state transformations and resource consumption in computations within a completely logical framework.

The starting point of this work is Forum [22, 23], a presentation of higher order linear logic which is an *abstract logic programming language*[24], i.e., complete with respect to *uniform proofs* (cut-free and goal-directed proofs). A subset of Forum's formulas which form the logic programming language $\mathcal{E}_{hhf}$[11, 13], have been isolated and proved to be well-suited to encode a notion of state into sequents and proofs, to model state updates, and to provide constructs for parallel execution.

In this paper, we will show how it is possible, using the language $\mathcal{E}_{hhf}$, to define a clean and simple semantics for object-oriented, deductive and active databases in a completely logical setting, that accounts for the various aspects of computation. In order to do this, we will briefly describe the semantics of Chimera [9, 19], an example of data model and language for DBMS which supports object-oriented, deductive, and active database features.

Keywords: higher order linear logic, abstract logic programming language, object-oriented, deductive and active databases.

1 Introduction

Linear logic [18] has acquired more and more importance in the last years, especially in the context of logic programming. Among the reasons for this success, there is the possibility of giving a natural encoding of notions such as state, events and resources, and of modeling state updates and concurrent computations in a completely *logical* setting. Linear logic, in fact, appears to be a logic of occurrences, in which a formula can be viewed as either an unlimited resource or as a limited one (this corresponds to the difference between unbounded and bounded context in sequent calculus); further, the possibility of modeling state updates in a logical way allows to overcome various problems related with traditional approaches (think about the primitives *assert* and *retract* in Prolog and the various update languages in deductive databases). Various languages have been developed; these include LO [1, 2, 3], ACL [21], Lolli [20], and Forum [22, 23]. Forum, in particular, is a presentation of full linear logic which is complete with respect to uniform proofs, i.e., cut-free and goal-directed ones. The language $\mathcal{E}_{hhf}$ represents an

attempt to isolate a subset of Forum's connectives which is both rich enough to model the notions reported above, and particularly simple to implement. A computation, in this context, can be viewed as proof construction (this is the so called *proof as computation interpretation* of linear logic).

In this paper we shall deal with applications of linear logic to databases. We will show how linear logic can give an account for several notions such as object-oriented data models (data can be naturally viewed as resources), deductive rules, and active rule management; this will allow us both to give a clear and simple semantics for such kinds of databases, and to define a rich programming language based on linear logic that can be used in this context. The presence of the $\parr$ connective in $\mathcal{E}_{hhf}$ makes it more suitable than other languages (see for example [20] for an application of Lolli to databases) to represent states and concurrent execution.

Current trends in database management systems (DBMS) include extensions to incorporate object modeling capabilities into deductive databases [4], or to incorporate triggers and constraints into object-oriented DBMS (OODBMS) [16, 17], or to enhance SQL with object-oriented capabilities [14]. A system, whose goal is specifically the integration of object-oriented, deductive and active capabilities has been developed as part of the ESPRIT Project Idea [6, 7]. In this work we shall describe the data model of Idea, called Chimera [9, 19, 8], and we shall give a semantics for it. We have chosen Chimera because it is a meaningful example of the integration of different paradigms, and, though not a commercial system, it is actually implemented.

It seems important here to base the semantics of such languages on a general purpose logic (Linear Logic) instead of studying ad hoc logical system (i.e. transaction logics to solve the problem of assigning a meaning to updates). Also, one of our future goals is to study all these formalisms in order to understand if they can be seen as subsystems of Linear Logic.

The interpretative approach sketched in this paper could also yield interesting results from the point-of-view of the design of specification languages based on logic languages like, for instance, Forum, and λProlog.

The rest of this paper is organized as follows: in section 2 we shall give a brief overview of $\mathcal{E}_{hhf}$, in section 3 we shall describe the main features of Chimera, and, finally, in section 4 we shall give a semantics for Chimera.

2 The language $\mathcal{E}_{hhf}$

The linear logic language $\mathcal{E}_{hhf}$[11, 13], standing for extended hereditary Harrop formulas, is a generalization of the system F&O[12] which has been defined to model state-based computations and in particular object-oriented aspects in Forum [22, 23].

For the sake of brevity, we shall not deal here with a formal proof theory (sequent calculus) for $\mathcal{E}_{hhf}$; instead, we shall limit ourselves to an informal presentation of the various rules. The reader can refer to [15] for a general presentation of sequent calculus, and to [11] for a detailed proof theory for $\mathcal{E}_{hhf}$.

The linear connectives considered in the language are $\parr$, &, $\multimap$, $\Rightarrow$, $\forall_\tau$ for each type τ, and the logical constants $\top$ and $\bot$. The two implications are written also $\circ\!\!-$ and $\Leftarrow$, usually to denote the top level implication of program clauses. The class of Forum formulas is restricted to two main classes, D- and G-formulas (D stands for *definite* clauses and G for *goals*):

$$
\begin{array}{lcl}
D & ::= & D \,\&\, D \mid \forall x.D \mid H \circ\!\!- G \mid D \Leftarrow G \mid H. \\
G & ::= & G \,\&\, G \mid G \parr G \mid \forall x.G \mid D \multimap G \mid D \Rightarrow G \mid A \mid \bot \mid \top. \\
H & ::= & H_1 \parr H_2 \mid A_r. \\
R & ::= & R \,\&\, R \mid R \parr R \mid \forall x.R \mid \bot \mid A_r.
\end{array}
$$

Here A represents a generic atomic formula, whereas A_r is a *rigid* atomic formula, i.e., whose top-level functor symbol is a constant. R-formulas are used to match the final result of a computation.

According to the *proof as computation interpretation* of linear logic, sequents represent the state of a computation. The left hand side of a sequent can be partitioned into two parts (Γ and Δ) consisting of the multi sets of unbounded and bounded resources, respectively; moreover, the right hand side of a sequent can be divided into two parts (Ω and Θ) which contain, respectively, processes to be executed (goals) and atoms representing the state of the computation (for instance a database state). Then sequents assume the following form:

$$\Gamma; \Delta[\Phi] \rightarrow_{\Sigma} \Omega \parallel \Theta,$$

where Γ and Δ are multi sets of D-formulas, Φ is an R-formula, Ω is a multi set of G-formulas, and Θ is a multi set of atomic formulas.

The rules of $\mathcal{E}_{hhf}$, defined in Appendix A, are divided into *right rules* (or *search rules*) and *left rules*. Right rules define the behavior of the various connectives: $\top$ is used to manage termination, $\bot$ to encode a null statement, & to split a computation into two branches which share the same resources, $\forall$ to encode a notion of *hiding*, $\parr$ to represent concurrent execution, $\Rightarrow$ and $\multimap$ to augment the resource context (respectively, the unbounded and the bounded context). The left rules define backchaining over clauses built with the connectives $\Leftarrow$ and $\circ\!\!-$. The rule for $\circ\!\!-$ is similar to Prolog rewriting, except that multiple-headed clauses are supported; besides, a clause can be reusable or not depending on which context it appears in. The rule for $\Leftarrow$ allows to depart an independent branch in an empty context (this is often useful to verify side conditions or make auxiliary operations).

In section 4, we shall see how the behavior of linear connectives can be successfully exploited to give a semantics for databases. For this purpose we will study the data model Chimera since it enjoys many interesting features of database languages as we will describe in the following section.

3 An overview of Chimera

The data model of Idea, called Chimera [8, 9, 19], is an object-oriented, deductive, active data model in that: it provides all concepts commonly ascribed to object-oriented data models (such as object identity, complex objects and user defined operations, classes, inheritance); it provides capabilities for defining deductive rules (used to define views and integrity constraints, to formulate queries, to specify methods, to compute derived information); finally, it supports a powerful language for defining triggers. In this section we shall briefly describe the main features of Chimera.

Objects and Values.
Chimera is a *class based* object oriented language for DBMS with *multiple inheritance*.

Objects are abstractions of real world entities, and are distinguished from each other by means of unique *object identifiers* (OIDs). The *state* of an object can be viewed as a collection of attributes (i.e., functions mapping an object to a uniquely defined value). Objects can be manipulated by means of *operations* (or *methods*), which are *guarded* procedures. An operation is defined by means of passive rules of the form `Head` ← `Body`. Objects are divided into *persistent* and *temporary* ones, depending on whether they survive a database session or not.

Classes.
The notion of *class* emphasizes the membership of objects in a common set of instances; classes are defined, populated, and deleted explicitly. *Class attributes* are functions mapping an entire class to a unique value. *Class operations* manipulate an entire class rather than individual instances (they can for example manipulate class attributes). Object classes may be recursively specialized into subclasses, resulting in a taxonomic hierarchy of arbitrary depth. A subclass inherits all attributes and operations from its super-classes, but may redefine their implementation and add new ones.

Example 3.1 *The real world entities 'person' and 'employee' can be modeled in Chimera by the following definitions:*

```
define object class person
  attributes      name: string(20)
                  income: integer
                  profession: string(10)
  operations      changeIncome (in Amount: integer)
  c-attributes    lifeExpectancy: integer
  c-operations    changeLifeExpectancy (in Delta: integer,
                    out NewValue: integer)
end

define object class employee
  super-classes   person
  attributes      emplNr: integer
                  mgr: employee
                  salary: integer
  c-attributes    maximumSalary: integer
end
```

Triggers.
Triggers or *active rules* are a means of introducing specific reactions to particular events relevant to the database. Such events include database specific operations (queries and updates), operation calls and constraint violations. The execution of reactions is subject to conditions on the database state reached whenever an event is monitored; further, triggers are prioritized. When one of the events of an active rule occurs, we say that the active rule is *triggered*. At given points of time, *active rule processing* is started: every triggered rule in considered and, if the condition associated with it is satisfied, the reaction is executed. Triggers can be defined as either *immediate* or *deferred*, depending on whether they are executed immediately or at the the end of the user transaction that has caused the triggering.

Example 3.2 *An example of trigger might be the following:*

```
define trigger adjustSalary for employee
  events      create
              modify (salary)
  condition   Self.salary > Self.mgr.salary
  actions     modify (employee.salary, Self, Self.mgr.salary)
end
```

Queries and Updates.
A *query* in Chimera consists of a formula F and a target list T. The formula F is evaluated over the current state of the database, returning bindings for the variables in T. *Updates* in Chimera support object creation and deletion, object migration from one class to another, state change or change of persistence status of objects. Finally, Chimera supports a conventional notion of *transaction*, with user-controlled *commit* and *rollback* primitives.

Example 3.3 *Typical queries in Chimera look like*

```
select (X.name where employee(X), X.salary < 5000)

select (X.name,Y.brand where person(X), car(Y), Y.owner=X).
```

4 A linear logic semantics for Chimera

In this section we will show how the various features of Chimera can be modeled using the language $\mathcal{E}_{hhf}$. We shall not deal explicitly with deductive rules, as, of course, this kind of rules are freely available in the context of a logical sequent calculus. All the details of the presented work can be found in [5]. In the following, we shall use the notation $\forall.F$ to indicate the universal closure of a formula F over all its free variables, and we shall write variables using identifiers in capital letters. The syntax of terms and formulas will be based on simply typed λ-calculus [10] (o will indicate the type of formulas).

Introduction.
We have chosen to model the *concurrent* execution of multiple user transactions, each of which has a sequential code. Therefore, the right hand side of sequents will contain an encoding of transactions together with a representation of the database state, while the left hand side will contain the unbounded resources, i.e. clauses, and the bounded ones, i.e. methods (as we shall see, a method must be activated before it can be used). The formulas we shall use in the following paragraphs for writing clauses have the general form

$$C_1 \ \& \ \ldots \ \& \ C_q \Rightarrow A_1 \parr \ldots \parr A_n \circ\!\!- B_1 \parr \ldots \parr B_m.$$

The informal meaning of such a formula is that, provided the conditions $C_1 \ldots C_q$ hold, the objects $A_1 \ldots A_n$ can be rewritten into $B_1 \ldots B_m$.

Representing computations.
We can think of a computation in a DBMS as the *parallel* execution of some user *transactions* submitted to the system. A transaction can be modeled by the term

$$\mathbf{trans}(code),$$

where *code* (the transaction's code) can be represented as a sequence of statements using the non-logical symbol $; : o \to o \to o$ (this technique is known as *continuations passing*). Hence, sequents will assume the following form:

$$\Gamma; \Delta[\Phi] \to_\Sigma \mathbf{trans}(code_1), \ldots, \mathbf{trans}(code_n) \parallel \Theta,$$

where, as said in section 2, Θ is a representation of the database state. The termination of a transaction can be managed by the following clause put in the unbounded context of a sequent:

$$\forall.\ \mathbf{trans}(\bot) \circ\!\!- \bot.$$

Object and class representation.
Let's now see how a hierarchy of classes can be modeled in our system (for the sake of simplicity, we shall limit ourselves to the case of single inheritance). We can represent the information on classes by clauses (in the unbounded context) of the form

$$class(clname, attrs, meths, supclname, extattrs),$$

where $attrs$ and $meths$ represent the attributes and methods of the class *clname*, while *supclname* is the superclass and $extattrs$ are the attributes which *clname* adds to those inherited from its superclass. The root class will be indicated as *topclass*.

The database state will be represented by means of a collection of object terms (appearing in the state part of a sequent) of the form

$$\mathbf{object}(clname, id, state),$$

where *id* is the *object identifier*, and *state* is a record of attributes.

Method representation.
A collection of methods can be modeled by a term like λID. $\text{METH}_1 \& \ldots \& \text{METH}_n$. A user transaction can invoke the execution of a method sending a message to an object, with a statement such as

$$\mathbf{send}(id, message),$$

where *message* specifies the name of the method and possibly some parameters. We can define the semantics of **send** by means of the following clause:

$$\begin{array}{l}\forall.\ class(\text{CLNAME}, \text{ATTRS}, \text{METHS}, \text{SUPCLNAME}, \text{EXTATTRS}) \Rightarrow \\ \quad \mathbf{trans}(\mathbf{send}(\text{ID}, \text{MESSAGE}); \text{R}) \parr \mathbf{object}(\text{CLNAME}, \text{ID}, \text{STATE}) \circ\!\!- \\ \quad\quad ((\text{METHS ID}) \multimap \mathbf{trans}(\mathbf{call}(\text{ID}, \text{MESSAGE}); \text{R}) \parr \mathbf{frozen}(\text{CLNAME}, \text{ID}, \text{STATE})).\end{array}$$

In the above clause, a mechanism of *object freezing* is used in order to prevent an object from responding to other method calls (this can be problematic because an object can migrate during its lifetime). A **send** primitive is then turned into a **call** primitive, which actually invokes method execution (the method code is activated in the bounded context). A method has the following base format:

$$\forall.\ \mathbf{trans}(\mathbf{call}(\text{ID}, message); r)\ \otimes\ \mathbf{frozen}(clname, \text{ID}, state) \circ\!\!- \mathbf{trans}(r'; r)\ \otimes\ \mathbf{object}(clname, \text{ID}, state')$$

where r' represents a sequence of statements added to the ones already present in r, and $state'$ is the result of updating the old state.

Example 4.1 *We could represent the relation between the classes person and employee (see example 3.1) by the clauses*

$$class(person, [name, income, profession], meths, topclass, []).$$
$$class(employee, [name, income, profession, emplNr, mgr, salary], meths', person, [emplNr, mgr, salary]).$$

For instance, the method changeIncome of the class person should appear something like

$$\lambda\text{ID}.\ \forall.\ set(\text{STATE}.income, income + Amount, \text{STATE}')\ \Rightarrow \mathbf{trans}(\mathbf{call}(\text{ID}, changeIncome(Amount)); \text{R})\ \otimes\ \mathbf{frozen}(person, \text{ID}, \text{STATE}) \circ\!\!- \mathbf{trans}(\text{R})\ \otimes\ \mathbf{object}(person, \text{ID}, \text{STATE}').$$

An object of type person could be, for instance, a term such as

$$\mathbf{object}(person, id, \langle name : Smith, income : 10000, profession : employee\rangle).$$

Object creation and deletion.
Object creation and deletion can be modeled by the following clauses (**iddaemon** represents a process, running in parallel with user transactions, that generates new identifiers for objects, and implemented as a simple counter):

$$\forall.\ class(\text{CLNAME}, \text{ATTRS}, \text{METHS}, \text{SUPCLNAME}, \text{EXTATTRS})\ \&\ buildrec(\text{ATTRS}, \text{VALUES}, \text{STATE})\ \Rightarrow \mathbf{trans}(\mathbf{create}(\text{CLNAME}, \text{VALUES}, \text{ID}); \text{R})\ \otimes\ \mathbf{iddaemon}(\text{ID}) \circ\!\!- \mathbf{trans}(\text{R})\ \otimes\ \mathbf{object}(\text{CLNAME}, \text{ID}, \text{STATE})\ \otimes\ \mathbf{iddaemon}(s(\text{ID})).$$

$$\forall.\quad \mathbf{trans}(\mathbf{delete}(\text{CLNAME}, \text{ID}); \text{R})\ \otimes\ \mathbf{object}(\text{CLNAME}, \text{ID}, \text{STATE}) \circ\!\!-\ \mathbf{trans}(\text{R}).$$

Queries.
For the sake of simplicity, we shall limit ourselves to queries on a single class, which can be written:

$$\mathbf{select}(attrlist\ \mathbf{intolist}\ l\ \mathbf{from}\ clname\ \mathbf{where}\ cond).$$

The informal meaning of this query is to select all the objects belonging to the class *clname* which satisfy the condition *cond*, and to retrieve into the list l the values of the attributes specified in *attrlist*. We will assume an auxiliary operation $get(object.attr, value)$ for retrieving the value of an attribute. The condition *cond* can be expressed in the following way:

$$\lambda\text{OBJ}.\lambda\overline{\text{ATTRS}}.cond_1\ \mathbf{and}\ \lambda\overline{\text{ATTRS}}.cond_2,$$

where COND_1 is an **and** conjunction of *get* predicates which retrieve the needed values of the attributes, while COND_2 is a condition over such values; $\overline{\text{ATTRS}}$ represents a tuple of variables.

Example 4.2 *The first query of the example 3.3 can be written like this:*

$$\textbf{select}([name]\ \textbf{intolist}\ l\ \textbf{from}\ employee\ \textbf{where}\ \lambda\textsc{obj}.\lambda\textsc{s}.get(\textsc{obj}.salary, S)\ \textbf{and}\ \lambda\textsc{s}.(\textsc{s} < 5000)).$$

Triggers.
Triggers have the form

$$(name, events, condition, actions),$$

and can be associated to a class definition. The predicate *class* can be modified adding a new parameter *trlist* which contains the list of triggers associated with the respective class. We can add a parameter to a user transaction, which becomes

$$\textbf{trans}(code, activetr),$$

where *activetr* is the list of activated triggers (actually, their code). The following rules manage trigger execution and transaction termination (we consider here *deferred* triggers):

$$\forall.\ \textbf{trans}(\bot, [\textsc{t}|\textsc{l}])\ \circ\!\!-\ \textbf{trans}(\textsc{t}, \textsc{l}).$$

$$\forall.\ \textbf{trans}(\bot, [])\ \circ\!\!-\ \bot.$$

All the clauses for the primitives of the DBMS must then be modified in order to take into account rule triggering. For instance, the clause for **modify** (the primitive for attribute modification) now becomes something like

$$\begin{array}{l}\forall.\ set(\textsc{state.attr}, \textsc{v}, \textsc{state}') \ \&\\ \quad checktrigg(\textsc{clname}, modify(\textsc{attr}), \textsc{t}')\ \&\\ \quad append(\textsc{t}, \textsc{t}', \textsc{t}'')\ \Rightarrow\\ \qquad \textbf{trans}(\textbf{modify}(\textsc{clname.attr}, \textsc{id}, \textsc{v}); \textsc{r}, \textsc{t})\ \otimes\ \textbf{object}(\textsc{clname}, \textsc{id}, \textsc{state})\ \circ\!\!-\\ \qquad\quad \textbf{trans}(\textsc{r}, \textsc{t}'')\ \otimes\ \textbf{object}(\textsc{clname}, \textsc{id}, \textsc{state}').\end{array}$$

A number of extensions for trigger management can be added with little effort; these extensions include *trigger inheritance*, *immediate triggers*, and *priority triggers*.

Other extensions.
Here, we shall briefly sketch the implementation of other features of Chimera that, for reason of space, we cannot discuss in details. *Class attributes* and *methods* can be managed by means of object terms like

$$\textbf{c\_object}(clname, state, meths)$$

in the state part of a sequent; method representation and calling can be simplified in this case. *Views* can be coded by means of clauses in the unbounded context of a sequent (they can be treated as predefined queries). Operations over lists of elements (which are very common in databases, and can be used to process query outputs) can be implemented in a straightforward manner. The notions of *persistent* and *temporary object* can be managed adding a new parameter $lifetime$ to the terms representing objects, and modifying the relevant rules. *Multiple inheritance* can be supported with little effort. User transactions can be enriched with some notion of *concurrency control*, which can be implemented using some form of *locking protocol*, and with *commit-rollback* primitives,

which can be implemented, as in real databases, with some kind of *log file*. Actually, the meta-level behavior of an interpreter for $\mathcal{E}_{hhf}$(through backtracking) already provides us with some notion of 'rollback', to some extent.

5 Conclusions and future work

In this paper, we have presented a *logical* framework which allows to model various aspects of database management systems: state updates, queries, object-oriented features, deductive rules, triggers, user transactions, and so on. Viewing a computation as a search for uniform proofs provides us with a clear *operational* semantics, and, what's more, the logical aspects of the framework we have used give us a well-founded *declarative* semantics.

The authors are persuaded that it would be very useful to build an effective implementation for the proof system presented here, so future work might include a λ-Prolog implementation for it. Moreover, future activity could address the problem of defining a general interpreter for the language $\mathcal{E}_{hhf}$, which appears to be very promising in the context of linear logic programming.

References

[1] J.M. Andreoli and R. Pareschi. Linear Objects: Logical Processes with Built-In Inheritance. In D.H. Warren and P.Szeredi, editors, *Proceedings of the 7th International Conference on Logic Programming*, pages 495–510. The MIT Press, Cambridge, MA, 1990.

[2] J.M. Andreoli and R. Pareschi. Communication as Fair Distribution of Knowledge. In *Proceedings of OOPSLA '91*, 1991.

[3] J.M. Andreoli and R. Pareschi. Linear Objects: Logical Processes with Built-In Inheritance. *New Generation Computing*, 9:445–473, 1991.

[4] E. Bertino, G. Guerrini, and D. Montesi. Deductive Object Databases. In M. Tokoro and R. Pareschi, editors, *Proceedings of the 8th European Conference on Object-Oriented Programming*, volume 821 of *Lecture Notes in Computer Science*, pages 213–235, 1994.

[5] M. Bozzano. Deductive databases in linear logic, 1997.

[6] S. Ceri and P. Fraternali. Draft of the idea methodology. Technical report, ESPRIT Rep.IDEA.DD.22P.001.02, 1994.

[7] S. Ceri and P. Fraternali. *Designing Database Applications with Objects and Rules: the IDEA Methodology*. Addison Wesley, 1997.

[8] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Active rule management in Chimera. In J. Widom and S. Ceri, editors, *Active Database Systems*. Morgan Kaufmann, 1994.

[9] S. Ceri and R. Manthey. Consolidated specification of Chimera. Technical report, ESPRIT Rep.IDEA.DE.2P.006.01, November 1993.

[10] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[11] G. Delzanno. *Logic & Object-Oriented Programming in Linear Logic*. PhD thesis, Università of Pisa, Dipartimento di Informatica, December 1996.

[12] G. Delzanno and M. Martelli. Objects in Forum. In *Proceedings of the International Logic Programming Symposium, Portland, Oregon*, pages 115–129. The MIT Press, 1995.

[13] G. Delzanno and M. Martelli. Proofs as computations. In *Proceedings of the GULP 96*, pages 115–129, 1995.

[14] L.J. Gallagher. Object SQL: Language Extensions for Object Data Management. In *Proceedings of the 1st International Conference on Information and Knowledge Management (CIKM)*, Baltimore, Maryland, November 1992.

[15] J.H. Gallier. *Logic for Computer Science*. Harper and Row, 1986.

[16] N. Gehani and H. Jagadish. Ode as an Active Database: Constraints and Triggers. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 327–336, 1991.

[17] N. Gehani, H. Jagadish, and O. Shmueli. Event Specification in Active Object-Oriented Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 81–90, 1992.

[18] J.Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1:1–102, 1987.

[19] G. Guerrini, E. Bertino, and R. Bal. A formal definition of the Chimera object-oriented data model, 1995.

[20] J. Hodas and D. Miller. Logic Programming in a Fragment of Intuitionistic Linear Logic. *Information and Computation*, 110(2):327–365, 1994.

[21] N. Kobayashi and A. Yonezawa. Asynchronous Communication Model based on Linear Logic. *Formal Aspects of Computing*, 7:113–149, 1995. Short version appeared in Joint International Conference and Symposium on Logic Programming, Washington, DC, November 1992, Workshop on Linear Logic and Logic Programming.

[22] D. Miller. A Multiple-Conclusion Meta-Logic. In *Proceedings of the 1994 Symposium on Logics in Computer Science, Paris*, pages 272–281, 1994.

[23] D. Miller. Forum: A Multiple-Conclusion Specification Logic. *Theoretical Computer Science*, 165(1):201–232, 1996.

[24] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform Proofs as a Foundation for Logic Programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

A Proof system

Search Rules

$$\frac{}{\Gamma;\Delta[\Phi]\rightarrow_\Sigma \top,\Omega \parallel \Theta}\ halt \qquad \frac{\Gamma;\Delta[\Phi]\rightarrow_\Sigma \Omega \parallel \Theta}{\Gamma;\Delta[\Phi]\rightarrow_\Sigma \bot,\Omega \parallel \Theta}\ erase \qquad \frac{\Gamma;\Delta[\Phi]\rightarrow_{y:\tau,\Sigma} A[y/x],\Omega \parallel \Theta}{\Gamma;\Delta[\Phi]\rightarrow_\Sigma \forall_\tau x.A,\Omega \parallel \Theta}\ hide\ (i)$$

$$\frac{\Gamma;\Delta[\Phi]\rightarrow_\Sigma A_1,\Omega \parallel \Theta \qquad \Gamma;\Delta[\Phi]\rightarrow_\Sigma A_2,\Omega \parallel \Theta}{\Gamma;\Delta[\Phi]\rightarrow_\Sigma A_1 \& A_2,\Omega \parallel \Theta}\ and$$

$$\frac{\Gamma;\Delta[\Phi]\rightarrow_\Sigma A_1,A_2,\Omega \parallel \Theta}{\Gamma;\Delta[\Phi]\rightarrow_\Sigma A_1 \parr A_2,\Omega \parallel \Theta}\ sync \qquad \frac{B,\Gamma;\Delta[\Phi]\rightarrow_\Sigma A,\Omega \parallel \Theta}{\Gamma;\Delta[\Phi]\rightarrow_\Sigma B \Rightarrow A,\Omega \parallel \Theta}\ augment \qquad \frac{\Gamma;B,\Delta[\Phi]\rightarrow_\Sigma A,\Omega \parallel \Theta}{\Gamma;\Delta[\Phi]\rightarrow_\Sigma B \multimap A,\Omega \parallel \Theta}\ fire$$

Backchaining

$$\frac{\Gamma;\Lambda\rightarrow_\Sigma \Omega \parallel \Xi}{\Gamma;\Delta\rightarrow_\Sigma \Upsilon \parallel \Theta}\ bc\ (ii)$$

Verify

$$\frac{}{\Gamma;\emptyset\,[\Phi]\rightarrow_\Sigma \emptyset \parallel \Theta}\ verify\ (iii)$$

Move

$$\frac{\alpha \in \Pi_{\Sigma_R} \quad \Gamma;\Delta\rightarrow_\Sigma \Omega \parallel \alpha,\Theta}{\Gamma;\Delta\rightarrow_\Sigma \alpha,\Omega \parallel \Theta}\ move$$

Context Rule

$$\frac{\Gamma;\emptyset\,[\Phi']\rightarrow_\Sigma G \parallel \emptyset \qquad \Gamma;\Lambda[\Phi]\rightarrow_\Sigma \Upsilon \parallel \Theta}{\Gamma;\Delta[\Phi]\rightarrow_\Sigma \Upsilon \parallel \Theta}\ empty\ (iv)$$

Given an H-formula $H = \parr_{i:1\ldots n} A_i$, let $H^\diamond$ denote the multiset $\{A_1,\ldots,A_n\}$, and $\langle D\rangle$ the set of instances of a clause D with implications as top level connective (i.e., $\langle D\&E\rangle = \langle D\rangle \cup \langle E\rangle$).

Definition A.1 (Subsystems) *A collection of subsystems of Υ, a multiset of atomic formulas, w.r.t. Γ and Δ, two multisets of D-formulas, is given by: the tuple $(\Delta_u,\Delta_b,\Delta_g)$, if $\Delta_u = \{D_1,\ldots,D_m\} \subseteq \Gamma$, $\Delta_b = \{D_{m+1},\ldots,D_n\} \subseteq \Delta$, $H_i \circ\!\!- B_i \in \langle D_i\rangle$, $i : 1\ldots n$; $\upsilon = \uplus_{i:1\ldots n} H_i^\diamond \subset \Upsilon$, and, Δ_g is the multiset $\{B_i \mid i : 1\ldots n\}$; the tuple $(\Delta_u,\Delta_b,\emptyset)$, if $\Delta_u \uplus \Delta_b \equiv \{D\}$, the H-formula $H \in \langle D\rangle$, and $\upsilon \equiv H^\diamond \equiv \Upsilon$.*

Definition A.2 (Multi-application resolvent) *Let $(\Delta_u,\Delta_b,\Delta_g)$ be a collection of subsystems of $\Upsilon \uplus \Theta$ w.r.t. Γ and Δ. If $\Delta_g \neq \emptyset$, then a multi-application resolvent of $(\Gamma,\Delta,\Upsilon,\Theta)$ is given by the tuple $(\Gamma,\Lambda,\Omega,\Xi)$ where $\Lambda = \Delta \setminus \Delta_b$, $\Omega = \Upsilon \setminus \upsilon$, $\Xi = \Theta \setminus \upsilon$. If $\Delta_g = \emptyset$, $(\Gamma,\Lambda,\Omega,\Xi)$ is a resolvent iff $\Lambda = \Delta \setminus \Delta_b$, $\Omega = \Upsilon \setminus \upsilon$, and $\Xi = \Theta \setminus \upsilon$ are all empty.*

A.1 Side conditions

The rules *bc*, $verify$, *empty* can be applied if and only if the right-hand side consists of atomic formulas, indicated by Υ; in the rule *move*, Π_{Σ_R} is a set of atomic state formulas. The side condition (i) of the *hide* rule requires that $y : \tau$ is not present in the signature Σ; (ii) requires $(\Gamma,\Lambda,\Omega,\Xi)$ to be a multi-application resolvent for $(\Gamma,\Delta,\Upsilon,\Theta)$ (if Λ, Ω and Ξ are empty, the *bc* reduce to an axiom scheme); (iii) requires $\Sigma : \Phi; \rightarrow \Theta$ to be provable in Forum. Finally, (iv) requires that $E \Leftarrow G \in \langle D\rangle$, $D \in \Delta$ and then $\Lambda = (\Delta \setminus \{D\}) \uplus \{E\}$, or $D \in \Gamma$ and then $\Lambda = \Delta \uplus \{E\}$.